Proteus—A microcoded multiprocessor system

G. M. Bull, S. Gilbey, Kathleen Levine, A. G. Lippiatt and C. H. C. Machin

Computer Systems Group, The Hatfield Polytechnic, P.O. Box 109, College Lane, Hatfield, Herts AL10 9AB

This paper describes the development of a microprogrammed, dual processor minicomputer system. The architecture of the special processor developed for the system is also described. The use of multiprogramming techniques with the microcode stored in a read/write memory together with a high speed context switching arrangement, gives rise to a powerful processor. A modular operating system based on a message/event system for a dual processor system is described.

(Received October 1976)

1. Introduction

Early in 1972 a team of staff in the Computer Science Department of The Hatfield Polytechnic decided to design and build a small computer system. The aims of the project team were:

1. To produce a minicomputer system which was particularly suitable for use in a teaching environment. The major characteristics required for teaching seemed to be:
   (*a*) the machine should be able to emulate other machines efficiently so as to give students experience of programming various types of machine
   (*b*) the machine should be capable of operating in a multi-program mode with adequate store protection so that students at later stages in their course could gain first hand experience at designing operating systems for such machines. It should also have features, such as segmentation and paging of the store, normally found on a larger scale machine, again so as to give students first hand experience of such systems.
2. To design a hardware system which would be cheap to reproduce, possibly on a small number of standard and/or custom built large scale integration chips. This would provide the necessary hardware for research into multiprocessor systems. Forecasts of what types of devices would be standard two or three years in the future were difficult to make, so the prototype is based on standard 74 series TTL devices which kept down the price whilst giving reasonable speed.
3. To give staff and students of the department first hand experience of the realisation of a computer system through all its stages from initial design discussion to the specification, design, building, commissioning and documentation of both the hardware and software. Such a project would provide a useful source of realistic projects for final year B.Sc. students and for students researching for a higher degree.

The software design is based around at least two processors. It was argued that a well designed multiprogramming operating system on a single processor should absorb less than 20% of the processor time so that a single 'operating system processor' should be able to control five or more 'user processors'. Since the operating system processor would have all the peripherals connected to it, it is called the input-output processor. It was further argued that if the user processors were to be mini-computers (characterised by a 16 bit word length and a limited instruction set/direct addressing range) it would be beneficial to use many of them to cooperate in the processing of user jobs. Thus one might have user processors acting as string processors, array processors, compiler processors, etc.

The input-output processor is a PDP11/10 with a disc and range of slow peripherals. The user processors are attached to the Unibus of the PDP11.

The hardware design team concentrated on the needs of the user processor. The resultant processor is called Proteus after the Greek god who was in the habit of appearing in different forms at different times. In the event Proteus exceeded the needs of the user processor as initially envisaged and now takes the form of a powerful, independent processor.

Proteus is provided with a 'soft' microprogramming system in which a random access read/write store is used to hold the microinstructions. It is also equipped with multiple register banks which include all the processor status information. A store protection and capability system are currently being designed.

In using Proteus as a user processor the instruction set (the first layer of virtual machine above the micro instruction set) can be changed when a process change occurs so that in practice all the different types of processor mentioned above can be realised on each user processor.

2. The hardware development of the Proteus processor

The hardware team were faced with a shortage of manpower, especially for construction, coupled with a shortage of finance for components. It was decided, therefore, to design as simple a processor as possible, but not to design out possible future enhancements. For example, the initial design does not incorporate a stack, but care has been taken to make sure that the design does not inhibit the later addition of a stack.

2.1 *Microprogramming*

In conventional computers, the lowest level at which the programmer can control the machine is at the machine code instruction set level. Each machine code instruction specifies a sequence of data transfers within the machine. The timing of the data transfer sequence (i.e. the sequencing) is performed by the machine control logic which also sets up the required data paths. Each of the data transfers can be called a microstep and the setting of the data path controls for each microstep can be considered to be a microinstruction. Hence a microinstruction specifies the setting of the data path controls for the microstep. A microprogrammed computer does not have a conventional instruction set, since it allows the programmer to specify the microinstructions—the setting of the data paths for each microstep—and it does not have control logic capable of executing only predetermined sequences of microsteps. The control logic allows execution of one microstep at a time. One would expect therefore that the instructions in a microprogrammed computer would be less powerful than the instructions in a conventional computer and would have to be executable in a shorter time than the instructions in a conventional machine to allow the machine to cope with the same workload. This is true if one takes the architecture of a conventional machine and tries to make it microprogrammed. However,

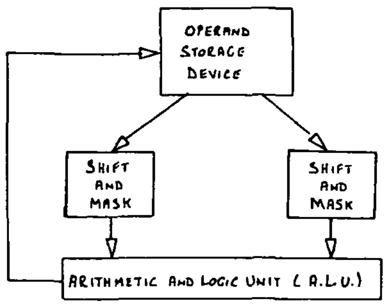


Fig. 1 Basic Proteus architecture.

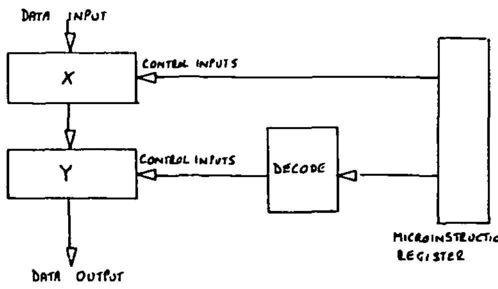


Fig. 2

if a microprogrammed machine is the design goal and if the architecture is chosen to suit microprogramming techniques, then it turns out that many of the microinstructions are very powerful, even compared with the instructions in conventional machines, as well as being executable very rapidly. In such a machine, and the processor being designed at The Hatfield Polytechnic is such a machine, the name microinstruction may be something of a misnomer, since the microinstruction set is in fact the machine instruction set and these instructions are very powerful. The basic architecture of the Proteus processor, as shown in **Fig. 1,** is very suitable for the application of microprogramming techniques.

For each microinstruction two operands are taken from the operand storage device, passed through a combinational logic system which operates on the operands, and the result is stored back in the operand storage device. This requires no sequencing because all the logic is combinational and each of the hardware blocks is equipped with control lines, the setting of which is specified by the programmer. In this machine, much of the power of the microinstructions is derived from the use of a large scale integrated Arithmetic Logic Unit (the 74181), which is provided with six control lines, and from the decision to include a combinational rather than sequential shift and mask unit in each input path to the Arithmetic Logic Unit. By shifting, using an integrated circuit multiplexor which is provided with an enable control, a shift of up to 16 places may be performed without waiting for the data to ripple through (a 16-to-1 multiplexor is used for each bit) and the enable control allows the output of the shifter to be forced to logic zero or to be the input data. Hence all shifts (except the arithmetic right shift) can be performed by a left rotational shift with suitable masking.

For example, a right shift of two places is the same as a left rotational shift of 14 places with masking of the two most significant bits of the result. This shift-and-mask logic allows any contiguous group of bits in the word to be output to the ALU. The group of bits may be zero to sixteen bits in length and at any position in the word.

2.2 *Choosing a microinstruction format*

The first difficult decision in the design of the microinstruction set is the choice of the number of bits in the microinstruction word. There are two extreme techniques. The first is to draw the complete block diagram of the machine showing all the control lines required for each block. The microinstruction will then contain one bit for each control line. All the controls in the machine can then be set up simultaneously and data will be transferred as determined by the control settings. This is called horizontal microprogramming. The second technique is to use a short microinstruction, say eight bits, and to go through a sequence of decoding steps until all the controls in the machine are set up. This is called vertical microprogramming. Usually horizontal microprogramming implies a one-for-one microinstruction bit for machine control correspondence and vertical microprogramming implies a high degree of encoding of information in the microinstruction.

The Proteus processor uses a microinstruction set which is a compromise between these extremes. Horizontal microprogramming provides great flexibility, but at the cost of a long microinstruction which has to be stored in an expensive high speed store. Intelligent examination of the significance of various controls in the machine indicates that considerable savings of microinstruction wordlength can be made without great sacrifice in flexibility of the code, because certain control combinations in the machine will be mutually exclusive; for instance, in the case when information is being written to a register, the setting of the (core) store controls is irrelevant, provided an initiate cycle signal is not sent.

So microinstructions specifying a register destination need not be capable of specifying a main store destination. Coding of control information in the microinstruction may save a large number of bits, but requires decoding. The main objection to this is the time required for the decode, but if the system is carefully chosen, the decoding time required may add nothing to the microinstruction execution time. For example, if two blocks X and Y (see **Fig. 2**) are controlled from the microinstruction register, provided the control signals for Y are decoded in less time than the data delay through X then no data delay will occur in Y due to control decoding.

In the Proteus processor, with the architecture shown in Fig. 1, the microinstruction contains coded information specifying the number of shifts, the mask format and the operand sources and destination. Decoding of the operand source is carried out by high speed logic, but decoding of the other information is carried out more slowly using slower logic. The mask is of the form: 0000 1111 1111 0000 where the 1s indicate which bits shall be allowed through to the ALU inputs.

The width of the block of 1s and their position in the word is specified in the microinstruction by giving, as binary numbers, the bit positions of the most significant 1 and the least significant 1 (bits 11 and 4 in the example above). This requires eight binary bits, rather than the 16 which would be required if the mask itself were specified in the microinstruction.

Again referring to Fig. 1, in Proteus the 'operand storage device' is, in fact, four separate types of device

(*a*) The registers

(*b*) The main store

(*c*) The control store (where the microinstructions are stored)

(*d*) The peripherals.

Since the architecture shown in Fig. 1 automatically requires a three address microinstruction (two operand sources and an operand destination) it was decided to provide sufficient hardware registers so that references to the main (slow) storage devices would be relatively infrequent and the most used microinstruction would then be required to specify the addresses of three registers plus the state of the controls in the combinational logic. Sixteen registers appeared to be sufficient.

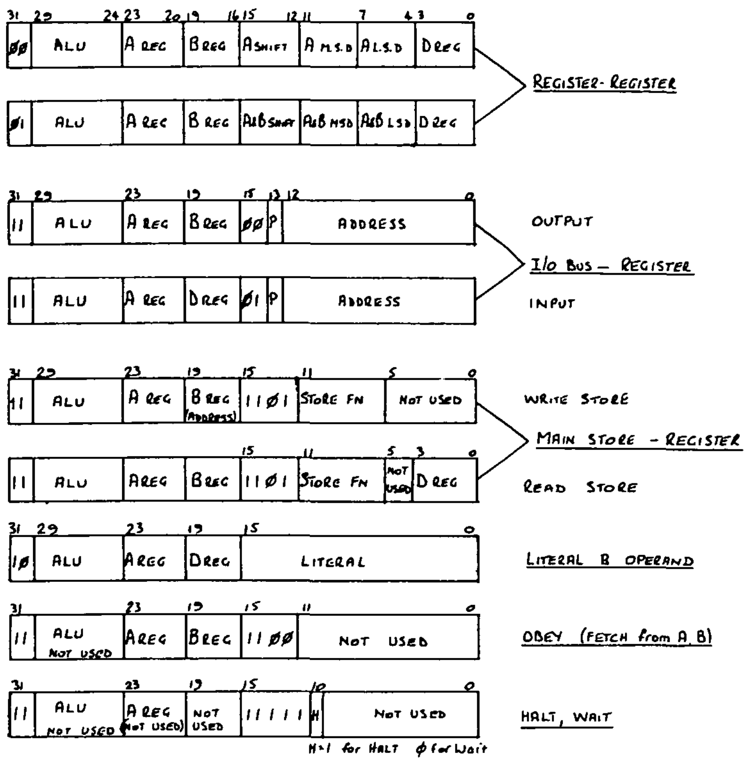


Fig. 3 Microinstruction set

The number of bits required in the microinstruction would then be:

(*a*) For addresses of three registers (A, B and D), 4 bits each	12 bits
(*b*) For the shifts, up to 15 places on A side and B side	8 bits
(*c*) For the mask, specify position of most significant bit and least significant bit required on each of A and B side	16 bits
(*d*) For ALU function	6 bits
A total of	42 bits

Allowing for the fact that there would be other microinstructions for addressing store a few bits will be required to specify the type of instruction, so it seems as if about 48 bits (3 × 16) are required in the microinstruction if a fixed length microinstruction format is used. Three word lengths is an awkward number so presumably two (32 bits) or four (64 bits) word lengths would be better.

Some speculative programs were written, which eventually led to the conclusion that a 32 bit microinstruction would give a reasonable compromise between cost, efficient use of storage and the power of the microinstruction set.

2.3 *The Proteus microinstruction set*

The complete set of microinstructions is shown in **Fig. 3.** These fall into four types.

(*a*) register-register (Modes 00, 01)

(*b*) register-control store and peripherals (Modes 1100, 1101)

(*c*) register-main (core) store (Mode 11 1101)

(*d*) miscellaneous, i.e. halt, literal load register, obey (various modes).

This division is a consequence of the architecture and the decision to use a 32 bit microinstruction which is too short to allow inclusion of more than one store address. The microinstruction format is described by the mode bits in the instruction.

It should be noted that in every case excepting Obey or Halt/Wait two operands pass through the shift and mask units and the ALU and data to the destination is always supplied by the ALU output. All instructions specify the ALU control settings and a register—the A register—which is the source of one operand.

The register-register instructions allow shifting and masking of the A register content only (mode 00) or of both the A and B registers (mode 01), before the ALU operation. The ALU output is deposited in the D register. These microinstructions are the most rapidly obeyed and in the prototype will take approximately 300 ns which is the worst case data delay through the system.

In register-control store instructions, the control store becomes the source of one operand or the operand destination. Instructions to read the control store cause the control store output to be substituted for the B register output and instructions to load the control store cause the ALU output to be directed to the control store. Under favourable circumstances in the machine these instructions take an average 450 ns (because the next microinstruction fetch is delayed). Register-core store instructions are similar, excepting that the core store address is held in one of the registers—the B register specified in the microinstruction—and data read from core is substituted for the B register output or data written to core is taken from the ALU output.

The time taken for these instructions is approximately 400 ns plus core store response time.

Since there are so few microinstruction formats, they are easily learned. The provision of a microassembler eases the task of coding programs.

2.4 *The Proteus processor architecture*

The basic architecture has been shown diagrammatically in Fig. 1 and used in developing the microinstruction set. The architecture will now be considered in more detail. This involves being rather more specific about the nature of the operand storage device of Fig. 1. This device has emerged as a mixture of registers, control store, peripherals, and main store. We shall consider each one in more detail.

2.4.1 *The Registers*

Sixteen 16 bit registers are provided (but see Section 2.5, Multiprogramming). All are addressable, some are general purpose and some are special purpose. The special purpose registers are:

(*a*) The Microprogram Counter—register 15

(*b*) The Processor Status Register—register 14.

The sixteen registers contain all the status information necessary for the running of a particular process.

The register set is provided with two output ports and an independent input port exactly as is indicated for the operand storage device of Fig. 1.

2.4.2 *The control store and peripherals*

This store is used as the principal store for microinstructions, hence it is 32 bits wide and is a read/write random access store. Since microinstructions should be retrievable from store at least as fast as they can be executed, and since in Proteus the register-register microinstruction takes about 300 ns then the control store requires a cycle time of the order of 200 ns maximum. Such store is expensive and has been realised using bipolar integrated circuits. The store is provided with two ports: one for the reading and writing of 16 bit words under the control of the register-control store microinstructions, and the other as an output port only for the output of 32 bit words directly to the microinstructions register. Since the register-control store microinstructions also address peripherals it is

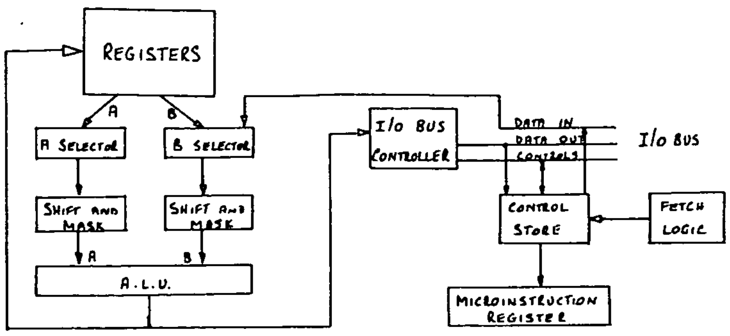


Fig. 4 Showing the control store position in the system

perhaps not surprising that the control store is connected to the machine I/O bus. The arrangement is shown in **Fig. 4** which also shows that there are two multiplexors (the A and B selectors) not previously mentioned, in the main data paths. These multiplexors are used to select the appropriate data source.

A write control store/peripheral microinstruction causes the I/O bus controller to send the address bits in the microinstruction along the I/O bus with a strobe at the appropriate time. This causes the receiving device (the control store or a peripheral device) to be activated in readiness to receive the data which are given to the I/O bus controller at the end of the microinstruction and sent by it early in the next microinstruction.

A read microinstruction causes a hold up in the processor until the I/O bus controller indicates that it has obtained the requested data.

The microinstructions are fetched by the control store at the request of the fetch logic, which determines the appropriate time in the machine cycle at which the fetch should be carried out. Normally the fetch will take place during the execution of the previous microinstruction, but there are exceptions, for example when the microinstruction is one which writes to the microprogram counter or to the control store.

2.4.3 *The main (core) store*

The main storage medium for immediate access is a core store, accessed by the Mode 111101 microinstructions shown in Fig. 3. The store is 16 bits wide and is designed as two interleaved stores each with a maximum 32K word capacity. The relation of the core store to the rest of the processor completes the processor block diagram which is shown in **Fig. 5**. The core store address is always taken from a register (the B register). If the core store is to be read, its data output is selected and passes through the B side of the processor to the ALU. If the core is being written, the ALU output is sent to its data input.

The core access controller can also deliver a 32 bit word directly to the microinstruction register. This allows microinstructions to be stored in the core store and to be obeyed directly from there. Although this is a relatively slow process, it will be useful for program debugging and for hardware-debugging programs. The fetch logic determines whether the required microinstruction is in core or in control store and issues the appropriate orders.

2.5 *Multiprogramming*

Proteus has been designed as a multiprogram machine. To facilitate rapid switching between processes, there is provision in the design for up to eight sets of 16 registers. Each set of registers is called a state vector (SV). Since each set of registers contains all the necessary information for the running of a process, process switching takes place very rapidly by switching between register blocks. Protection of registers is achieved by the use of a privilege flag in the processor status register. If this flag is set, registers belonging to other processes may be accessed by addressing them as the top 128 peripheral registers. Only privileged processes may access the peripherals. By using more than one flag (there are presently unused flags available) a graded privilege structure could be introduced. Four state vectors are provided in the prototype.

Although not provided in the first prototype, an early enhancement will be the provision of hardware main and control store protection. Some protection is obtainable by carrying out core store address checking in microcode.

2.6 *Interrupts*

As the microinstruction is in fact the machine level instruction, interrupts will be recognised by the hardware at the end of every microinstruction. Recognition of an interrupt will cause a state vector (process) switch. Since each interrupt level will be able to specify which state vector to switch to, high priority interrupts will receive very rapid service if the operating system arranges for the service routine to be ready loaded in the appropriate state vector and control store.

2.7 *Direct memory access*

Direct memory access for peripheral devices is a possible later enhancement, but because the processor is capable of high speed context switching and the microinstruction time is short, it is estimated that the processor will be capable of handling data transfers of about 1 million 16 bit words per second under program control. DMA facilities will, therefore, be required only in larger scale installations than that currently envisaged.

3. Software development for the Proteus system

The Proteus processor provides a means of implementing existing designs of instruction sets and of experimenting with new ones. The microprogram used to realise a given instruction set may also be used to collect statistics on programs, for research into programming techniques, and to provide tracing and debugging facilities not present in the actual processor being emulated.

The operating system design, as noted earlier, is based on a multiprocessor system with one of the processors (the input/output processor) controlling one or more user processors. This approach has of course had a considerable influence on our thinking. Thus even though the Proteus processor has evolved as a two state machine with peripheral channels, and capable of being operated in a conventional manner, we have concentrated our efforts on a dual input/output processor/user processor configuration.

3.1 *Microprogram development aids*

The microprogram assembler at present being used was produced by adding a set of macros to an existing macro assembler (Macro-10 on the DEC system10). Each microinstruction format has a macro whose parameters are the fields within the microinstruction. A call of one of these macros produces a word in the output of the assembler containing the microinstruction as it is to appear in control store on the microprogrammed processor. This assembler has proved adequate so far, mainly because of the small size of microprograms. All the facilities of the powerful host macro assembler are made available for very little effort in implementing the special macros. The main disadvantages are that error checking during assembly is poor and the assembly listing is sometimes difficult to interpret.

To help with the development, a simulator for Proteus has been developed. It provides all the facilities of the hardware, together with the ability to examine store locations and set breakpoints in a microprogram.

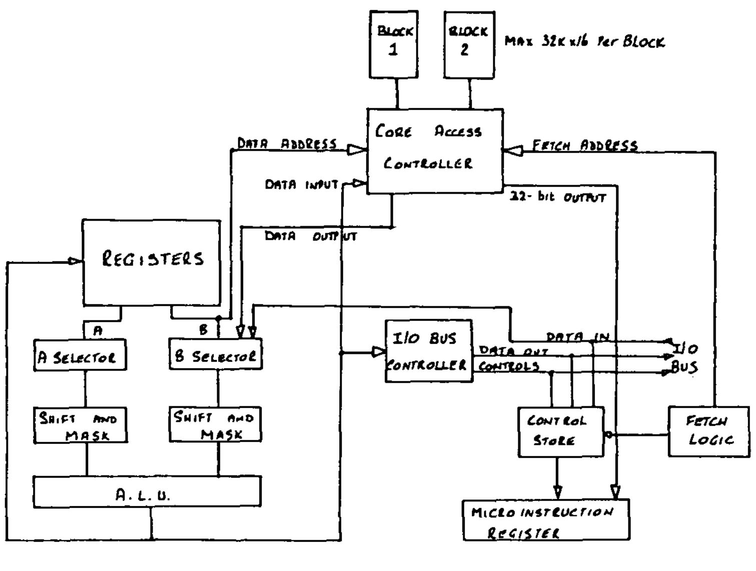


Fig. 5 Proteus processor block diagram

3.2 *Language implementation*

There is currently much interest in systems in which the design of the overall system and the design of the instruction set of the processor are considered together. The Proteus processor allows experiments along these lines. Currently a compiler for the POP-2 language is being designed which will make use of instruction sets designed specifically for it. POP-2 was chosen because it is an interesting language which is difficult to implement efficiently using conventional instruction sets. The compiler is being designed to produce the object code using one of several different instruction sets, depending on the local characteristics of the program being compiled. The information about the program will be supplied by the programmer on the basis of expected and measured performance. The instruction sets will range from designs having low store requirements to designs having fast execution rates. It is expected that typical programs will have significantly reduced space and time requirements without the extended compilation time needed by an optimising compiler with similar performance.

3.3 *Operating systems*

One of the design goals of the software team was to produce a system which could be used in teaching operating systems. The major requirements of such a system are that it can be used as a model for the detailed study of a particular operating system design, and that it can provide the basis for practical work in operating systems. In a relatively short course on operating systems, it is clear that no student has the time to write a complete operating system; it is doubtful if a group of students working together could achieve such a goal. So that students do not waste their time on irrelevant details, it is necessary to:

(*a*) provide a fully documented system

(*b*) have a standard, well defined interface between processes

(*c*) write the system in a high level language.

With such provisions, students may be asked to redesign or replace a given process or set of processes or add new processes so that they fully understand the function of the processes in question and their overall role within the system. With the above in mind, a dual processor operating system has been designed as a set of asynchronous co-operating parallel processes, which may communicate via a message/event system running in the input/output processor.

The kernel of the operating system is the message handler and is responsible for the scheduling of system processes and for the creation and deletion of all processes in the system. **Fig. 6** shows the overall structure of the operating system.

The system processes fall into the categories of management of storage, files, input/output, job control, supervisor call (SVC) handling, user process scheduling and measurement; some of these categories will include a number of processes.

The message system provides processes with a message buffer into which information to be transmitted, for example a request from one system process for service from another, can be written together with some identification of the receiver and of the sender. As the message is 'transmitted' by the movement of message buffer pointers to a queue associated with the receiver, a priority is attached to the message to allow some degree of system process scheduling. The sender may immediately, or at some time in the future, wait for an answer to come back or may perhaps wait for a message to arrive from some other process.

The event system is an extension of the message system which allows a numbeı of processes to be activated on the occurrence of an event for which they have been waiting. For example, some activities may generate processes to handle them, each of which request service from some other process. Each of the subprocesses generated to handle the initial activities may need to know the outcome of the same terminal process (for example swapping device operation to free some core store). The event system would allow a 'broadcast' of this information to the processes awaiting it.

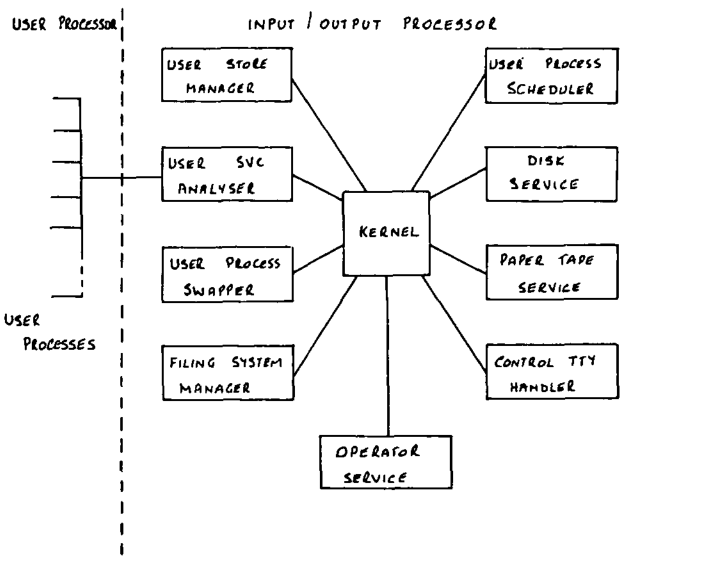


Fig. 6 Logical interrelationships between representative modules of the operating system, the kernal and the user processes

$$\begin{tikzcd}
\text{Enter with state vector number} \arrow[d] & & \\
\text{Is it an SVC?} \arrow[r, "\text{No}"] \arrow[d, "\text{Yes}"] & \text{Has user process elected to service this error?} \arrow[r, "\text{No}"] \arrow[d, "\text{Yes}"] & \text{Tell kernel to run standard error handling process E} \arrow[d] \\
\text{Identify service process and tell kernel to run it} \arrow[d] & \text{Tell kernel to run process U} \arrow[d] & \text{Return to kernel} \\
\text{Return to kernel} & \text{Return to kernel} &
\end{tikzcd}$$

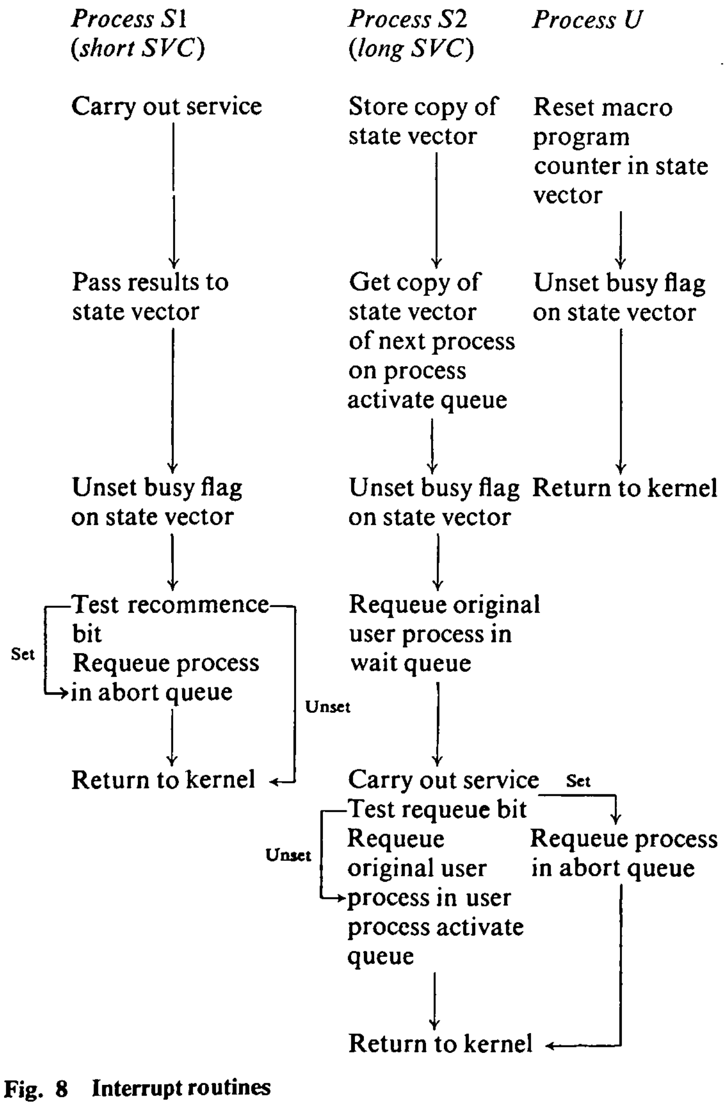
Fig. 7 User processor interrupt analysis

3.3.1 *Interprocessor communications*

The user process scheduler is activated as a result of either a user process requesting an operating system service by means of an SVC or as a result of a user process fault condition, e.g. exceeded time allocation, overflow. When a user process executes an SVC, a bit corresponding to the state vector number is set in a register and an interrupt is generated to the input/output processor. On generating this interrupt the state vector associated with the process which generated the interrupt is marked as busy and a scan is made by the state vector selection mechanism for a non-busy state vector. If one is found the user processor is associated with this state vector and activates the corresponding process. If no non-busy state vector is found the user process hangs until one is made non-busy by the input/output processor. When the user process generates a fault condition, the condition is set in the processor status register and an interrupt is generated to the input/output processor as in the case of an SVC.

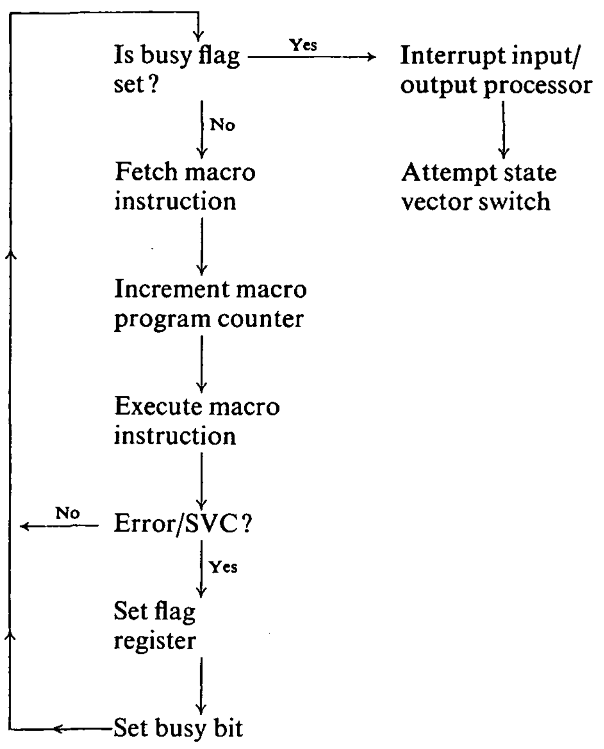
The input/output processor response to an interrupt from the user processor is shown in **Fig. 7.** It first checks the processor status register and if there is a fault condition indicated, deals with it appropriately. One of the SVC's available to the user process enables it to indicate that it would like to deal with certain fault conditions and to indicate the name of the routine to deal with them. If no fault condition is indicated then an SVC is presumed and the appropriate action taken.

When the input/output processor comes to deal with an SVC the normal action is to: (*a*) store the contents of the corres-



*Process S*1 (*short SVC*)
*Process S*2 (*long SVC*)
Process U
Carry out service
Pass results to state vector
Unset busy flag on state vector
Test recommence bit
Set
Requeue process in abort queue
Unset
Return to kernel
Store copy of state vector
Get copy of state vector of next process on process activate queue
Unset busy flag on state vector
Requeue original user process in wait queue
Carry out service
Set
Test requeue bit
Unset
Requeue original user process in user process activate queue
Requeue process in abort queue
Return to kernel
Reset macro program counter in state vector
Unset busy flag on state vector
Return to kernel

Fig. 8 Interrupt routines



Is busy flag set?
Yes
Interrupt input/output processor
Attempt state vector switch
No
Fetch macro instruction
Increment macro program counter
Execute macro instruction
No
Error/SVC?
Yes
Set flag register
Set busy bit

Fig. 9 User processor macro instruction cycle

ponding state vector in the core store of the input/output processor, (*b*) mark the appropriate system process which will deal with the SVC as requiring the input/output processor, (*c*) reload the state vector with the status information of the user process on the front of the user process activate queue, (*d*) clear the busy flag for that state vector, and (*e*) dismiss the interrupt.

There is however a class of SVC for which the service routine execution time is small in comparison with the time to unload and load a state vector. An example of such an SVC is one which asks for the current time of day. The action taken in this case is to leave the state vector busy until the 'result' of the system process for that SVC can be loaded directly into the state vector. The busy flag for that state vector is then cleared and the interrupt dismissed.

The action following an interrupt from the user processor to the input/output processor is shown in **Fig. 8.**

The input/output processor may wish to abort a running or runnable user process at any time. Such a user process may be in one of four states:

(*a*) Runnable but waiting to be loaded into a state vector. The process is removed from the user process activate queue

(*b*) Loaded in a state vector awaiting activation. The busy flag of the state vector is set and the system process to deal with this situation is requested to run

(*c*) Running in a user processor. The busy flag of the state vector is set. This will be detected on the next macro (emulated) instruction cycle as shown in **Fig. 9**

(*d*) Running as a system process in the input/output processor as a result of (say) an SVC. The requeue bit for the user process is set. When the system process running on behalf of the user process completes, it tests the requeue bit and if it finds it set, requeues the user process in the abort queue.

Future developments

The Proteus project offers the opportunity for many exciting developments. In the first instance, a Proteus processor will be used to replace the initial input/output processor (a PDP11). No changes to the operating system software will be required since Proteus will simply emulate the PDP11. However, when this is undertaken, no doubt some or all of the kernel will be recoded directly into microcode to improve the efficiency. Other projects under active consideration are the use of a simple version of Proteus as a general purpose peripheral controller, as a communication multiplexor, as a computer network interface and as an interprocessor communication unit. Further, the investigation of multiple processor systems in a variety of configurations may be undertaken when the large scale integration of Proteus has been undertaken. For example, a single input/output processor controlling a number of user processors will be investigated as will the interconnection of a number of such systems in both a hierarchical and a distributed network.

Bibliography

Gaines, R. S. (1972). An operating system based on the concept of a supervisory computer, *CACM*, Vol. 15, No. 3.

Hansen, P. B. (1970). The Nucleus of a Multiprogramming system, *CACM*, Vol. 13, No. 4.

Wilner, W. T. (1972). B1700 Memory Utilisation, *AFIPS FJCC.*

DTSS System Reference Documents. Dartmouth College, USA.

Digital Equipment Corporation. PDP11 processor handbook.

Book review

Current trends in programming methodology: Volume I.—Software specification and design, Edited by Raymond T. Yeh, 1977; 275 pages. (*Prentice-Hall*, £13·55)

Standardised development of computer software, by Robert C. Tausworthe, 1977; 379 pages. (*Prentice-Hall*, £15·95)

Two books, from the same publisher, with ostensibly similar subject matter, yet written and produced in apparent ignorance of each other. For example both have reasonable bibliographies yet neither mentions the other. Examination also shows that the contents of these bibliographies are not at all similar: some references appear in both, but these are limited to such fundamentals as Donald Knuth on *The Art of Computer Programming*.

This situation demonstrates something of the dichotomy which has appeared in publications concerned with programming. On the one hand we have the academics and research workers who are attempting (and often succeeding) to penetrate the mystery surrounding programming and to determine how the various tasks may be performed more accurately and more productively. On the other hand we have those involved in programming, especially in a user environment, and providing or applying various tools for better control of the activity.

It has often been said that the two parties are quite separate and have little to say to each other. Some DP people, especially, claim to see little advantage in some of the ideas propounded and suggest that a pseudo-scientific aura is being created around what is properly a craft. They further claim that the methods suggested by the academics have been known and practised for years. Sometimes one may read academics who pour scorn on DP practices stating that they have little in the way of theoretical justification. I was once told, for example, that NCC's Programming Techniques manual was 'too pragmatic' and therefore of little account.

It is a pity that this dichotomy exists for each camp has much to discuss with the other. Recognition that it exists is, however, useful in interpreting the raison d'etre of the two books under review.

In his preface, Raymond T. Yeh states the intention of the series as 'to bring together a collection of tutorial papers . . . which are representative of the current trends' (in programming methodology). This first volume surveys recent developments concerned with 'the systematic design of well structured and reliable software architecture'. And so we have nine papers in all from: Liskov and Zilles; Wulf; Robinson; Levitt; Neumann and Saxena; Parnas; Linger and Mills; Knuth; Randell; Naur; Dijkstra. Six of these papers have appeared elsewhere we are told, two of them having since been updated. The material is all of recent date, however, and this is to be applauded. One grows a little tired of being served the same old fare, however re-hashed.

Yeh concludes his preface by suggesting the book can be used by 'upper-division' undergraduates or first year graduates in computer science but, rightly, points out that it should be supplemented by exercises and projects. He also suggests it should act as a reference book for 'professional software engineers'.

For the first group there is little doubt that this is an excellent text book. The preface itself provides a good explanation of first principles and demonstrates how the rest of the book hangs together. It is also supported by a 214-entry bibliography, which is well annotated, and the index steers a middle course between identifying only fundamental points in the text and listing every passing reference made. One might quibble, not so much with what is included but with what is omitted. Mills on 'How to write correct programs and know it' and Wirth's 'On the composition of well structured programs' should be required reading for any student wishing to be informed. Both are, of course, well described in the bibliography.

Whether or not the second group is so well catered for is a matter of debate. What, in *British* terms, is a 'professional software engineer'? The Americans tend to use the term 'software' to mean everything which is programmed whereas we tend to constrain this to mean operating systems, compilers and everything that goes with the machine, i.e. not applications. If we take the British usage then, yes, the book should be of value to anyone wishing to explore the theories of program structure. But in all probability the average applications programmer will find much of the content hard to read and understand. Such a person would do better to read Tausworthe's book.

Robert C. Tausworthe is of the Jet Propulsion Laboratory at the California Institute of Technology, Pasadena. His book (or 'monograph' as he calls it) is based upon standards produced to guide the work of a team producing a conversational version of a BASIC compiler. These standards evolved naturally over a period of time, and much of the book appeared originally as material for computer science courses at West Coast University and JPL seminars.

The ten chapters examine a number of fundamentals concerning program structure and behaviour. These are then used in describing design, development and testing methods in the context of applied programming. The whole is related, not only to programming practice but also to the organisation and management of that practice. Each chapter concludes with a summary and sets a number of problems to enable the reader to try out what he or she has just 'learned': the use of the book as a training aid is therefore obvious.

Anyone who has tried to keep an eye on programming methodology will recognise much of what Tausworthe suggests we do. Hierarchies, step-wise refinement, machine independent design approach, high level design specification language, team organisation, walkthroughs, use of a librarian: these and others are all ideas which appear in the book though not necessarily in those terms. It would be easy to dismiss the book as 'nothing particularly new'.

But suppose you *are* an applications programmer, or someone who runs such people? Suppose you have not had the time to read or absorb much of the literature, or indeed are confused by it all? This book would help you. It may not be the *best* book but you will find it useful, particularly if you want to get down to basics and not worry too much about philosophic argument. I cannot suggest what

Continued on page 167